CS 423
Operating System Design:
"Virtual" Machines
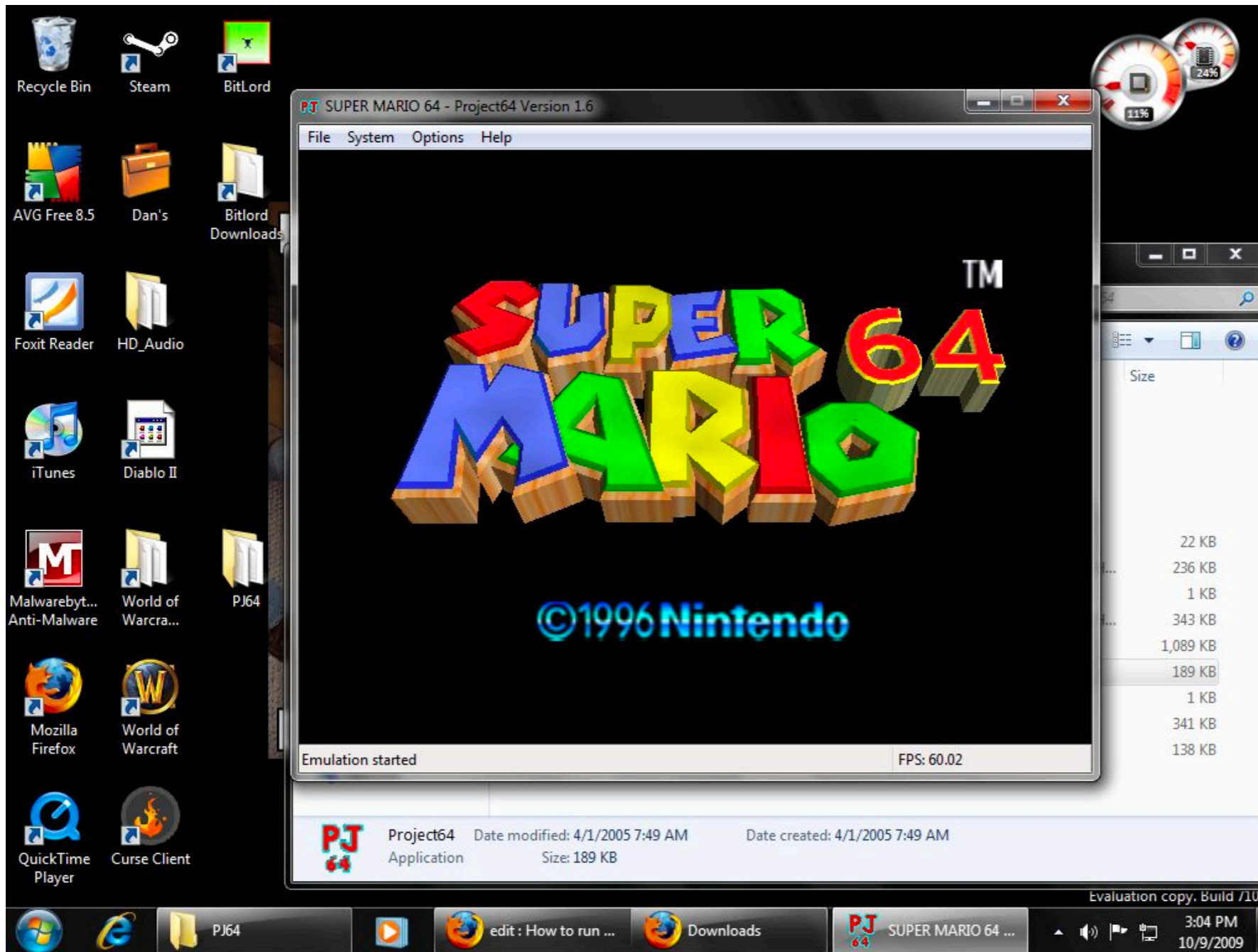
Tianyin Xu

- The OS has thus far served as the illusionist, tricking unsuspecting applications into thinking they have their own private CPU and a large virtual memory, while secretly switching between applications and sharing memory.

- Why do we need another level of indirection (virtualization)?

Yet another level of virtualization?

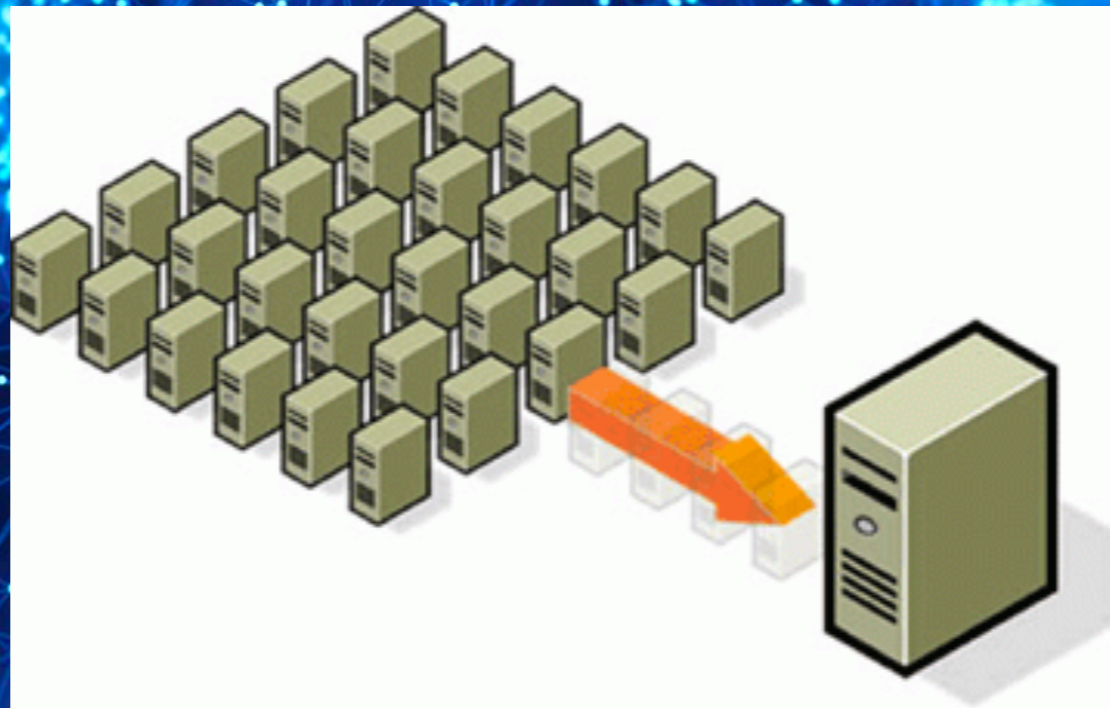# Yet another level of virtualization?

Containerization vs Virtualization

- What's the difference from containers and virtual machines?

- How about chroot, jails, and zones?

- What is the difference between Xen and VMWare ESX?

- What are they virtualizing?
  - VM
  - JVM
  - LLVM

Virtualization

- Creation of an isomorphism that maps a virtual guest system to a real host:
  - Maps guest state S to host state V(S)
  - For any sequence of operations on the guest that changes guest state S1 to S2, there is a sequence of operations on the host that maps state V(S1) to V(S2)

Important Interfaces

- ## Application programmer interface (API):
  - High-level language library such as `libc`

- ## Application binary interface (ABI):
  - User instructions (User ISA)
  - System calls

- ## Hardware-software interface:
  - Instruction set architecture (ISA)

What's a machine?

- Machine is an entity that provides an interface
  - From the perspective of a language…
    - Machine = Entity that provides the API
  - From the perspective of a process…
    - Machine = Entity that provides the ABI
  - From the perspective of an operating system…
    - Machine = Entity that provides the ISA

What's a virtual machine?

- Virtual machine is an entity that emulates a guest interface on top of a host machine
  - Language view:
    - Virtual machine = Entity that emulates an API (e.g., JAVA) on top of another
    - Virtualizing software = compiler/interpreter
  - Process view:
    - Machine = Entity that emulates an ABI on top of another
    - Virtualizing software = runtime
  - Operating system view:
    - Machine = Entity that emulates an ISA
    - Virtualizing software = virtual machine monitor (VMM)

Purpose of a VM

- Emulation
  – Create the illusion of having one type of machine on top of another

- Replication (/ Multiplexing)
  – Create the illusion of multiple independent smaller guest machines on top of one host machine (e.g., for security/isolation, or scalability/sharing)

- Optimization
  – Optimize a generic guest interface for one type of host

Types of VMs

- Emulate (ISA/ABI/API) for purposes of (Emulation/Replication/Optimization) on top of (the same/different) one.

Types of VMs

- Emulate (ISA/ABI/API) for purposes of (Emulation/Replication/Optimization) on top of (the same/different) one.
  - Process/language virtual machines (emulate ABI/API)
  - System virtual machines (emulate ISA)

Types of VMs

- Emulate (ISA/ABI/API) for purposes of (Emulation/Replication/Optimization) on top of (the same/different) one.
  - Process/language virtual machines (emulate ABI/API)
  - System virtual machines (emulate ISA)

- Emulate what interface?

- For what purpose?

- On top of what?

Ex1: Emulation

- **Emulate one ABI on top of another** (early emulation wants to run Windows apps on MacOS)
  - Emulate an Intel IA-32 running Windows on top of PowerPC running MacOS (i.e., run a process compiled for IA-32/Windows on PowerPC/MacOS)
    - Interpreters: Pick one guest instruction at a time, update (simulated) host state using a set of host instructions
    - Binary translation: Do the translation in one step, not one line at a time. Run the translated binary

Writing an Emulator

- Create a simulator data structure to represent:
  - Guest memory
    - Guest stack
    - Guest heap
  - Guest registers

- Inspect each binary instruction (machine instruction or system call)
  - Update the data structures to reflect the effect of the instruction

- Emulate one ABI on top of itself for purposes of optimization
  - Run the process binary, collect profiling data, then implement it more efficiently on top of the same machine/OS interface.

Ex3: Language VMs

- Emulate one API on top of a set of different ABIs
  - Compile guest API to intermediate form (e.g., JAVA source to JAVA bytecode)
  - Interpret the bytecode on top of different host ABIs

- Examples:
  - JAVA
  - Microsoft Common Language Infrastructure (CLI), the foundation of .NET

Types of VMs

- Emulate (ISA/ABI/API) for purposes of (Emulation/Replication/Optimization) on top of (the same/different) one.
  - Process/language virtual machines (emulate ABI/API)
  - System virtual machines (emulate ISA)

- Emulate (ISA/ABI/API) for purposes of (Emulation/Replication/Optimization) on top of (the same/different) one.
  – Process/language virtual machines (emulate ABI/API)
  – System virtual machines (emulate ISA)

System VMs

- Implement VMM (ISA emulation) on bare hardware
  - Efficient
  - Must wipe out current operating system to install
  - Must support drivers for VMM

- Implement VMM on top of a host OS (Hosted VM)
  - Less efficient
  - Easy to install on top of host OS
  - Leverages host OS drivers

System VMs

- Implement VMM (ISA emulation) on bare hardware **TYPE ONE HYPERVISOR**
  - Efficient
  - Must wipe out current operating system to install
  - Must support drivers for VMM

- Implement VMM on top of a host OS (Hosted VM) **TYPE TWO HYPERVISOR**
  - Less efficient
  - Easy to install on top of host OS
  - Leverages host OS drivers

What is Xen?

What is VirtualBox?

What is KVM/Qemu?
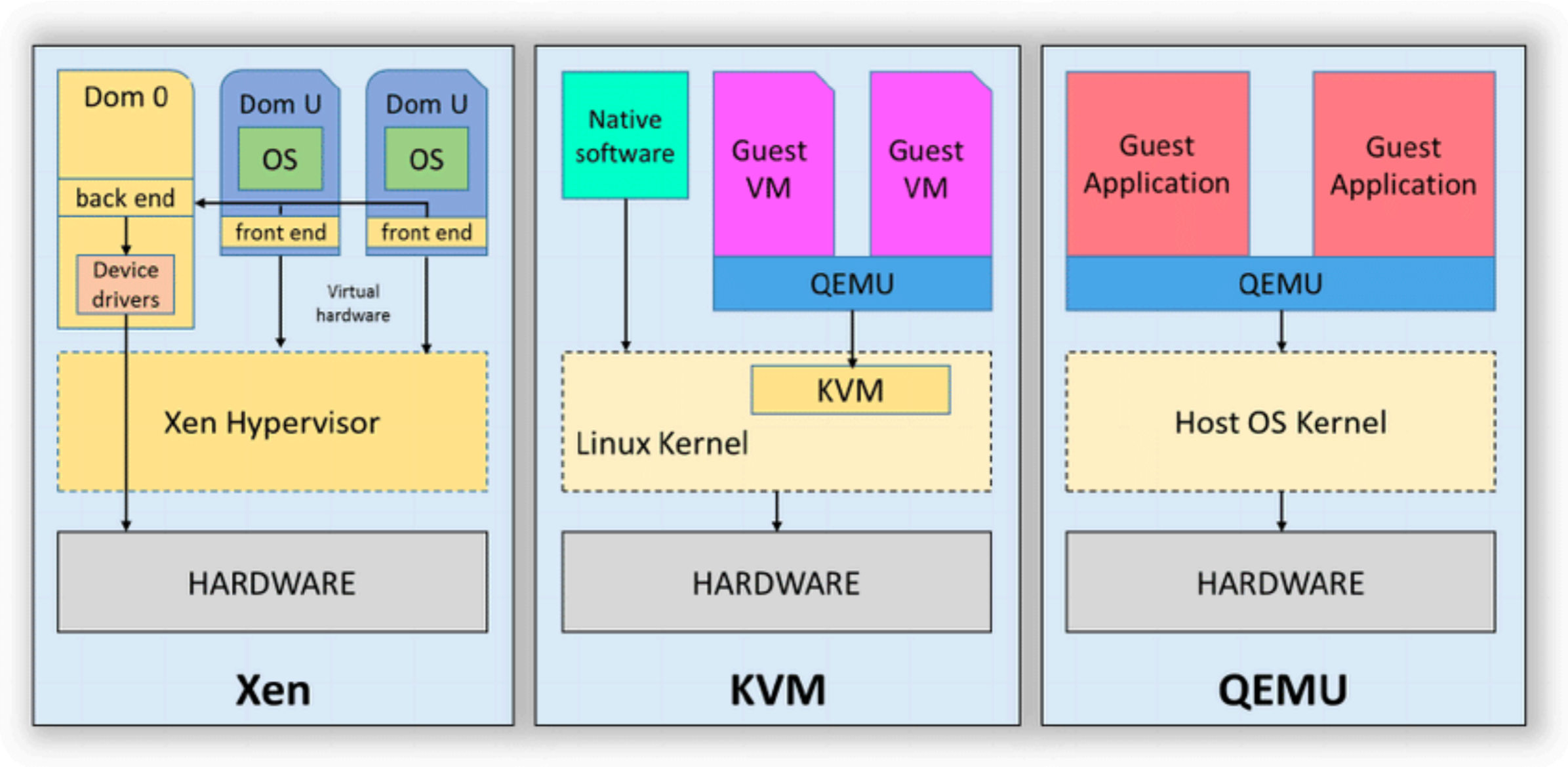
- Language VMs
  - Emulate same API as host (e.g., application profiling?)
  - Emulate different API than host (e.g., Java API)

- Process VMs
  - Emulate same ABI as host (e.g., multiprogramming)
  - Emulate different ABI than host (e.g., Java VM, MAME)

- System VMs
  - Emulate same ISA as host (e.g., KVM, VBox, Xen)
  - Emulate different ISA than host (e.g., MULTICS simulator)

Point of Clarification

- Emulation: General technique for performing any kind of virtualization (API/ABI/ISA)

- Not to be confused with *Emulator* in the colloquial sense (e.g., Video Game Emulator), which often refers to ABI emulation.

- Problem: Emulate guest ISA on host ISA

Writing an Emulator

- Problem: Emulate guest ISA on host ISA

- Create a simulator data structure to represent:
  - Guest memory
    - Guest stack
    - Guest heap
  - Guest registers

- Inspect each binary instruction (machine instruction or system call)
  - Update the data structures to reflect the effect of the instruction

Emulation

- Problem: Emulate guest ISA on host ISA

- Solution: Basic Interpretation, switch on opcode

```
inst = code (PC)
opcode = extract_opcode (inst)
switch (opcode) {
        case opcode1 : call emulate_opcode1 ()
        case opcode2 : call emulate_opcode2 ()
    …
}
```

Emulation

- Problem: Emulate guest ISA on host ISA

- Solution: Basic Interpretation

```
new              inst = code (PC)

                 opcode = extract_opcode (inst)

                 routineCase = dispatch (opcode)

                 jump routineCase

                 …

routineCase                  call routine_address

                             jump new
```

[body of emulate_opcode1]

inst = code (PC)

opcode = extract_opcode (inst)

routine_address = dispatch (opcode)

jump routine_address

[body of emulate_opcode2]

inst = code (PC)

opcode = extract_opcode (inst)

routine_address = dispatch (opcode)

jump routine_address

Emulation

- Problem: Emulate guest ISA on host ISA

- Solution: Basic Interpretation, switch on opcode

```
inst = code (PC)

opcode = extract_opcode (inst)

switch (opcode) {

    case opcode1 : call emulate_opcode1 ()

    case opcode2 : call emulate_opcode2 ()

    …

}
```

Emulation

- Problem: Emulate guest ISA on host ISA

- Solution: Basic Interpretation

```
new              inst = code (PC)

                 opcode = extract_opcode (inst)

                 routineCase = dispatch (opcode)

                 jump routineCase

                 …

routineCase                     call routine_address

                                jump new
```

[body of emulate_opcode1]

inst = code (PC)

opcode = extract_opcode (inst)

routine_address = dispatch (opcode)

jump routine_address

[body of emulate_opcode2]

inst = code (PC)

opcode = extract_opcode (inst)

routine_address = dispatch (opcode)
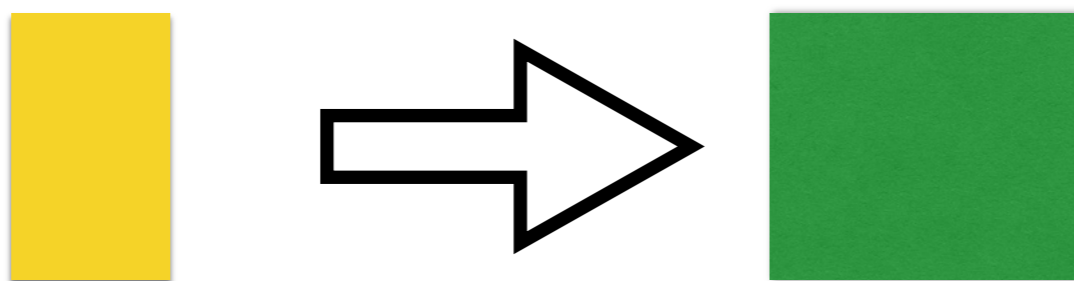
jump routine_address

- extract_opcode (inst)
  - Opcode may have options
  - Instruction must extract and combine several bit ranges in the machine word
  - Operands must also be extracted from other bit ranges

- Pre-decoding
  - Pre-extract the opcodes and operands for all instructions in program.
  - Put them on byte boundaries…

  - Also, must maintain two program counters. Why?

  Source Code   Intermediate Code

Example: MIPS Instruction Set

0x1000: LW   r1, 8(r2)

0x1004: ADD   r3, r3, r1

0x1008: SW   r3, 0(r4)

135		
1	2	08

0x10000: LW

032		
3	1	03

0x10008: ADD

142		
3	4	00
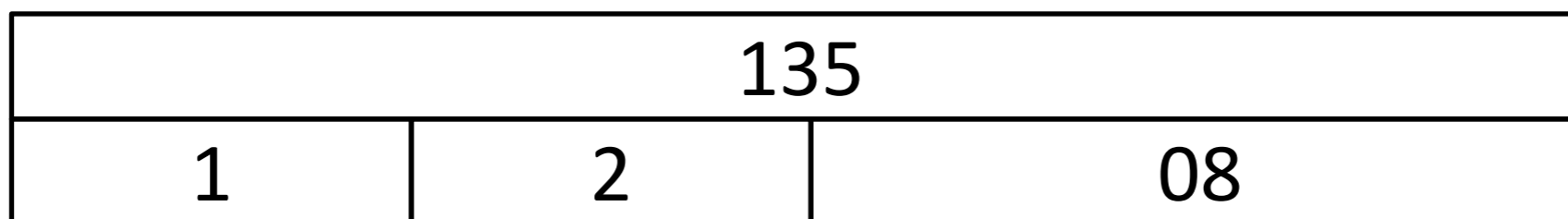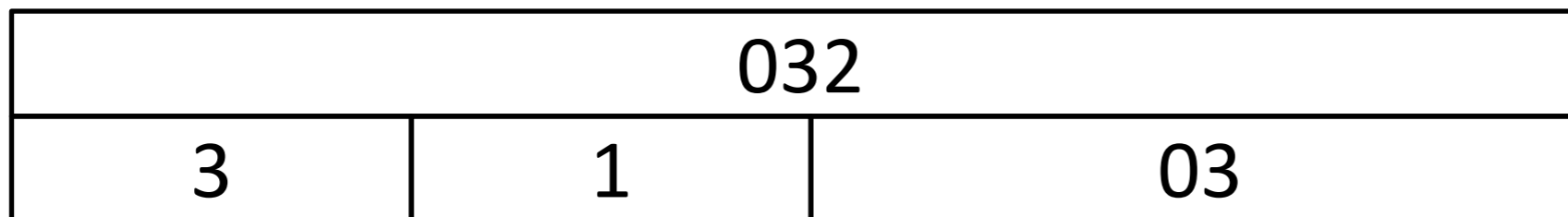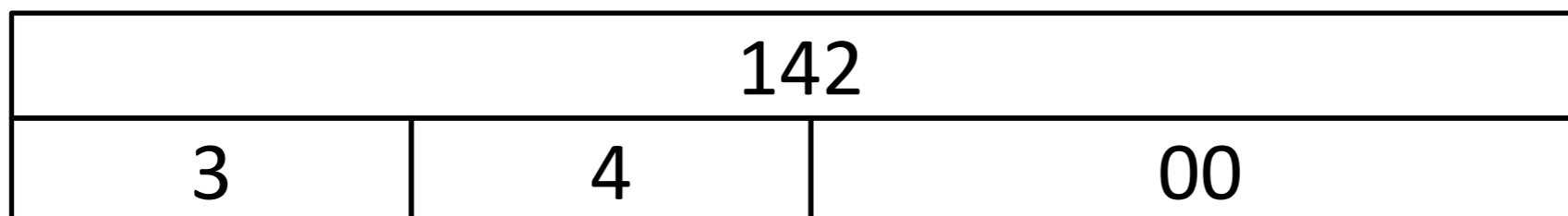
0x10010: SW

- Replace opcode with address of emulating routine

Routine_address07		
1	2	08

Routine_address08		
3	1	03

Routine_address37		
3	4	00

Binary Translation

- Emulation:
  - Guest code is traversed and instruction classes are mapped to routines that emulate them on the target architecture.

- Binary translation:
  - The entire program is translated into a binary of another architecture.
  - Each binary source instruction is emulated by some binary target instructions.

Challenges

- Can we really just read the source binary and translate it statically one instruction at a time to a target binary?
    - What are some difficulties?

Challenges

- Code discovery and binary translation
  - How to tell whether something is code or data?
  - We encounter a jump instruction: Is word after the jump instruction code or data?

- Code location problem
  - How to map source program counter to target program counter?
  - Can we do this without having a table as long as the program for instruction-by-instruction mapping?

- You only need source-to-target program counter mapping for locations that are *targets of jumps*. Hence, only map those locations.

- You always know that something is an instruction (not data) in the source binary if the source program counter eventually ends up pointing to it.

- The problem is: You do not know targets of jumps (and what the program counter will end up pointing to) at static analysis time!
 – Why?

Solution

- Incremental Pre-decoding and Translation
  - As you execute a source binary block, translate it into a target binary block (this way you know you are translating valid instructions)
  - Whenever you jump:
    - If you jump to a new location: start a new target binary block, record the mapping between source program counter and target program counter in map table.
    - If you jump to a location already in the map table, get the target program counter from the table
  - Jumps must go through an emulation manager. Blocks are translated (the first time only) then executed directly thereafter

Dynamic Basic Blocks

- Program is translated into chunks called "dynamic basic blocks", each composed of straight machine code of the target architecture
  - Block starts immediately after a jump instruction in the source binary
  - Block ends when a jump occurs
- At the end of each block (i.e., at jumps), emulation manager is called to inspect jump destination and transfer control to the right block with help of map table (or create a new block and map table entry, if map miss)

Dynamic Binary Translation
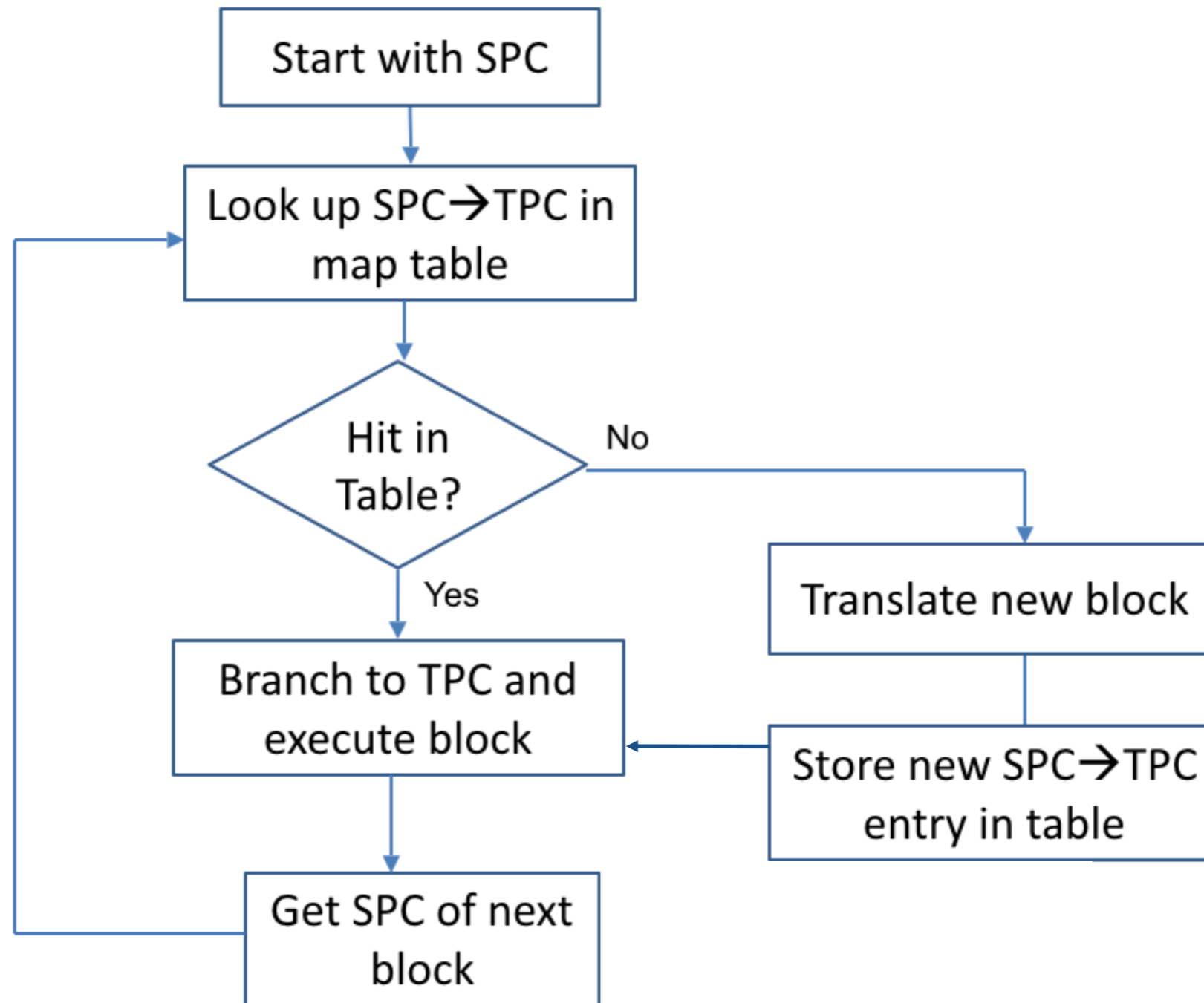
Edit: The original automata didn't execute the current block unless there was a hit!

Optimizations

- Translation chaining
  - The counterpart of threading in interpreters
  - The first time a jump is taken to a new destination, go through the emulation manager as usual
  - Subsequently, rather than going through the emulation manager at that jump (i.e., once destination block is known), just go to the right place.
    - What type of jumps can we do this with?

Optimizations

- Translation chaining
  - The counterpart of threading in interpreters
  - The first time a jump is taken to a new destination, go through the emulation manager as usual
  - Subsequently, rather than going through the emulation manager at that jump (i.e., once destination block is known), just go to the right place.
    - What type of jumps can we do this with?
      - Fixed Destination Jumps Only!!!

Register Indirect Jumps?

- Jump destination depends on value in register.

- Must search map table for destination value (expensive operation)

- Solution?
  - Caching: add a series of if statements, comparing register content to common jump source program counter values from past execution (most common first).
  - If there is a match, jump to corresponding target program counter location.
  - Else, go to emulation manager.